MSc thesis

Master's Programme in Computer Science

Program Equivalence Checking for Automatic Recognition of Quantum-Compatible Code

Jon Speer

October 1, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Jukka K. Nurminen

Examiner(s)

Prof. Tommi Mikkonen

Prof. Jukka K. Nurminen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Jon Speer

Työn nimi — Arbetets titel — Title

Program Equivalence Checking for Automatic Recognition of Quantum-Compatible Code

Ohjaajat — Handledare — Supervisors

Prof. Jukka K. Nurminen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	October 1, 2020	63 pages

Tiivistelmä — Referat — Abstract

The techniques used to program quantum computers are somewhat crude. As quantum computing progresses and becomes mainstream, a more efficient method of programming these devices would be beneficial. We propose a method that applies today's programming techniques to quantum computing, with program equivalence checking used to discern between code suited for execution on a conventional computer and a quantum computer. This process involves determining a quantum algorithm's implementation using a programming language. This so-called benchmark implementation can be checked against code written by a programmer, with semantic equivalence between the two implying the programmer's code should be executed on a quantum computer instead of a conventional computer. Using a novel compiler optimization verification tool named CORK, we test for semantic equivalence between a portion of Shor's algorithm (representing the benchmark implementation) and various modified versions of this code (representing the arbitrary code written by a programmer). Some of the modified versions are intended to be semantically equivalent to the benchmark while others semantically inequivalent. Our testing shows that CORK is able to correctly determine semantic equivalence or semantic inequivalence in a majority of cases.

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords

Quantum Computing, Program Equivalence Checking, Shor's Algorithm

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

In this thesis, we analyze the use of program equivalence checking for the recognition of code suitable for execution on a quantum computer. The goal is to propose a method that can be applied at some point in the future where general-purpose quantum computers are no longer a theoretical construct, but rather an integral part of the computing landscape. We consider the following research questions:

1. What methods can be used to automatically identify code suitable for execution on a quantum computer?
2. What are the limitations of these methods' capabilities?

A quantum computer is a device that exploits the bizarre nature of quantum mechanics to deliver significant performance gains relative to "conventional" computers for particular types of problems. (The term "conventional computer" refers to the CMOS-based architecture found in today's computers.) We can say that a quantum computer is comprised of quantum hardware and the software that runs on it. The theory behind these machines has been in existence for several decades [8] [10], with actual working quantum computers (however primitive they may be) having been built by various organizations [14] [7].

Achieving the full potential of quantum computing will require not only the requisite quantum hardware, but also the corresponding software. Such a massive change in computing architecture would seemingly necessitate an equally drastic change in the software running on this new type of hardware. However, abstraction is a fundamental component of programming language design and computing in general. While the "machine code" running on a quantum computer will obviously need to comply with the underlying architecture, code written by the developer need not be significantly different from present-day programming languages. The key to abstracting away the hardware is the compilation process, much like today.

Considering the beginning stage at which quantum computing is currently at, it is not surprising that the software side of quantum computing is equally primitive. At present-day, there exist several publicly-available online toolkits [22] [20], allowing for a user to create a series of instructions that are then executed on a quantum computer "in the cloud". Some of these offerings involve simulated quantum hardware, while others, such

as IBM's Qiskit [22], use real quantum hardware to run a user's code. Programming the quantum computer is done by manually manipulating quantum gates or calling APIs.

A programming language should minimize the knowledge required of the underlying hardware, i.e. abstract away the hardware. This should rule out gate-level manipulation in a future where quantum computing is ubiquitous. In general, Qiskit is what we should expect of the current state of quantum programming. It would be unreasonable to expect that the methods for quantum software development be much more advanced than its hardware counterpart. Qiskit is fine for today's state of quantum computing, but if we are to realize a future where quantum computers are as mainstream as their conventional counterparts, then it is necessary to consider a more practical method of quantum software development.

This thesis envisions a future in which quantum computers have achieved non-trivial capabilities that greatly surpass the limitations faced by conventional computers. Realization of this capability means that *certain* types of tasks will be completed much more quickly on a quantum computer than on a conventional computer, while other types of tasks will see *no* speedup on a quantum computer. Thus, the latter may be best left to conventional computers. It is reasonable to assume that both computing technologies will exist in parallel, with quantum computers "in the cloud" and conventional computers found locally, i.e. in laptops, cell phones, etc. This stems not only from the aforementioned limitations of quantum computers, but also from the cost and complexity of quantum computers. While it is likely that production costs will fall as technology progresses, conventional computers will likely remain cheaper to produce - possibly significantly cheaper - meaning that keeping the more expensive quantum computers in the cloud will be an economically preferable approach. The much larger physical size of quantum computers (as of the writing of this thesis) also makes the cloud approach preferable.

Assuming this cloud-based model, and a computing landscape where programmers have a need for quantum hardware on a somewhat regular basis (we will expand on this point later), code will be separated into conventional and quantum categories, meaning that programmers will need to write code for both types of architectures. While dedicated languages already exist for the programming of quantum computers, the gradual incorporation of the quantum realm into mainstream programming languages would make sense from a practicality standpoint. One approach is the use of APIs: programmers would have libraries at their disposal to perform tasks on quantum hardware. However, this would require a programmer to have a requisite understanding of these APIs; by no means insur-

mountable, considering that APIs are commonplace today. However, it's worth considering if another approach is feasible, one where the programmer need not even think about the underlying hardware.

This alternative approach is to abstract away the conventional and quantum realms altogether from the programmer's perspective. Instead of the programmer knowing, or even caring, about what should be executed on a conventional computer and what should be executed on a quantum computer, a compiler or interpreter (i.e. translator) would make that decision. This concept is the focus of this thesis.

1.1 Quantum Code Recognition

Determining whether code should be executed on a quantum computer or conventional computer is based upon recognizing algorithms that are known to exhibit improved performance when executed on a quantum computer. It is up to the translator to make the determination of whether or not code should be executed on a conventional target or a quantum target.

Our algorithm recognition process is as follows. We start with an algorithm that we want executed on quantum hardware and its corresponding quantum implementation via a combination of quantum logic gates. With this in hand, we need to determine the algorithm's implementation using a programming language. This implementation serves as the benchmark we can use to compare against any arbitrary block of code and determine if they are semantically equivalent (semantic equivalence implies that the "meaning" of the programs is the same, even if written differently). Thus, we have a method that links a quantum algorithm and its implementation on a quantum computer to code written by a programmer. Our hypothetical translator would be tasked with identifying this algorithm during the process of translation, resulting in this particular block of code being executed on a quantum computer, while "non-quantum" code would be executed on a conventional computer.

Now we will elaborate on this process. First, we address the correlation between the quantum algorithm and the benchmark. Consider a quantum algorithm, such as Shor's algorithm for integer factorization. The implementation of Shor's algorithm on a quantum computer must be determined (i.e. in terms of quantum logic gates). The next step is to determine the implementation of Shor's algorithm via a programming language. With this in hand, we now have a link between the algorithm's quantum executable and its

programming language equivalent (the benchmark). The more difficult part is recognizing equivalent code with respect to the benchmark. It is up to our hypothetical translator to perform program equivalence checking between the code it translates and the various benchmark implementations of quantum algorithms. See Figure 1.1.

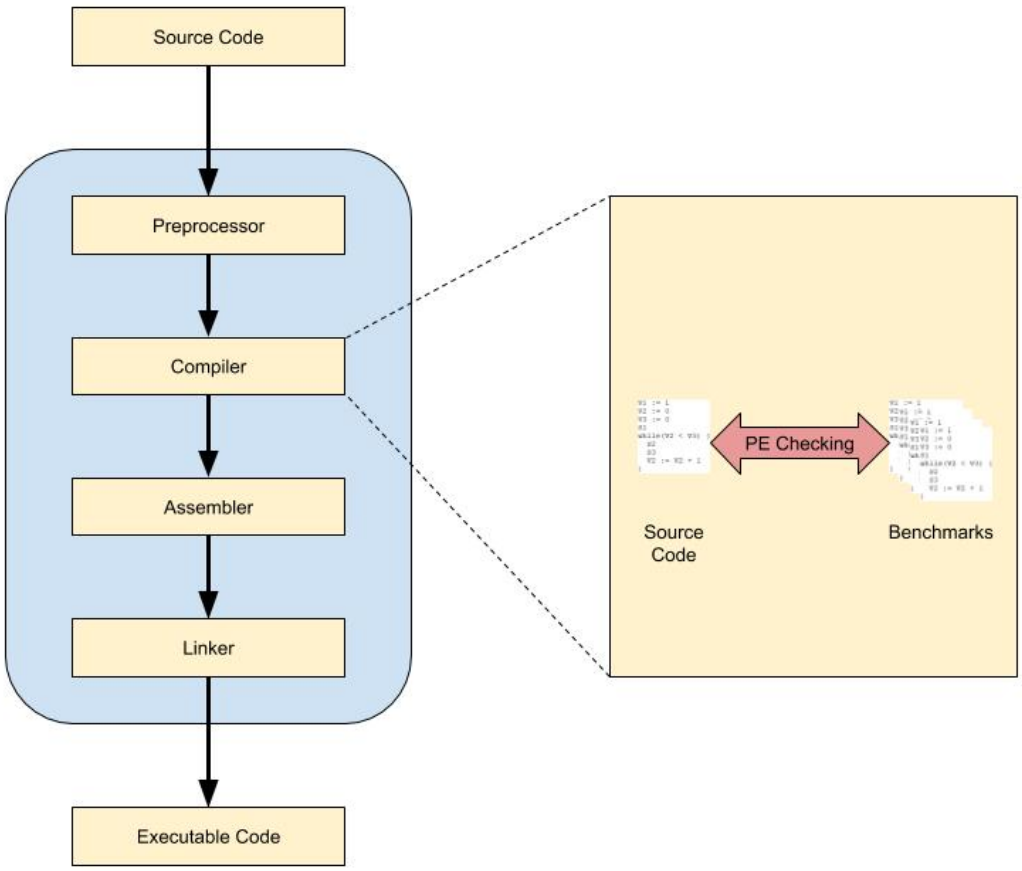

Figure 1.1: Depiction of proposed program equivalence checking process as implemented in a C compiler. The right-hand side depicts source code being checked against the collection of benchmarks, corresponding to various quantum algorithms. Not shown are necessary compilation steps, such as lexical analysis, syntax analysis, semantic analysis, etc.

Figure 1.2 illustrates our proposed program equivalence checking method, with a quantum algorithm implemented via some programming language (the C programming language in this example). This benchmark is the translator's "gold standard" that is used to compare against the code it translates. The translation process would thus entail the translator analyzing source code against the benchmark. As might be expected, there would likely be

more than one benchmark to check against, considering that each benchmark corresponds to a single quantum algorithm.

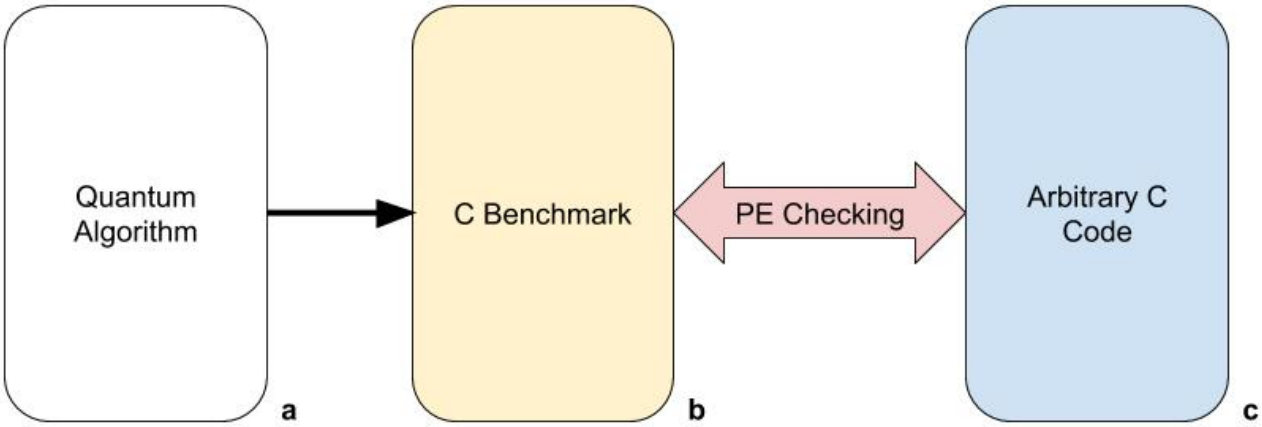

Figure 1.2: The hypothetical program equivalence checking procedure proposed by this thesis. A quantum algorithm (**a**) is implemented via some programming language (the C programming language in this example) and serves as the benchmark (**b**). Program equivalence checking is performed by our hypothetical translator (Figure 1.1) between the benchmark and any arbitrary C code (**c**). Semantic equivalence implies that this code is an implementation of a quantum algorithm and should thus be executed on a quantum computer.

Referring back to our Shor's algorithm example, if our hypothetical translator finds a semantically equivalent implementation during compilation, this algorithm is compiled for a quantum target instead of a conventional target (we will discuss this in more detail later). In this sense, what happens "under the hood" is similar to the API model discussed earlier in that execution of this particular code occurs on a quantum computer instead of a conventional computer. The difference is that the API model relies on the programmer to explicitly make this determination, whereas our proposal entails an automatic approach.

Program equivalence has present-day applications, among them being the verification of compiler optimizations [5] [17] and algorithm recognition [2]. This in turn relates to the focus of this thesis: the analysis of code with the goal of finding conventional equivalents of quantum algorithms. Our proposal therefore has a base of knowledge to build upon.

1.2 Potential for Future Usage

In evaluating the potential for this concept to become a useful tool at some point in the future, the first thing to consider is the potential of quantum computing itself. Though still

in its infancy, the field is rapidly gaining attention and interest. Whether this momentum continues will be determined by the progress made by research into quantum computing.

Assuming quantum computing does indeed realize its currently understood potential (or even exceeds it), we then need to consider how these devices will be programmed. Implementing the idea discussed in this thesis would require a shift toward a programming environment seen today, with the programming languages used for programming conventional computers also used for programming quantum computers. It is not too much of a stretch to foresee a future where, like previously discussed, performing a task on a quantum computer is a matter of calling the right API. Yet another alternative would be today's situation, where programming a quantum computer involves a dedicated language. The API model seems like a more practical alternative, and would thus present the biggest challenger to our proposal.

The question becomes, does our proposal present a practical alternative to the API model? The use of APIs has been in existence for quite some time. Expanding this concept to include quantum computing seems reasonable, if not entirely likely. The drawbacks associated with utilizing APIs include the requisite knowledge a programmer must possess in order to use the API to its full potential. If a programmer doesn't know that a quantum library function exists for a particular algorithm, he or she might implement its conventional equivalent instead, resulting in potentially longer execution time. Though this in turn is dependent upon the range of problems for which a quantum computer can provide a performance increase, i.e. how large the "quantum API" is. The smaller the API, the easier it is for programmers to be familiar with all of it. As research into quantum computing progresses, a clearer picture of the range of problems well-suited for quantum computers will be known. An alternative to the API model becomes more realistic as this range of problems grows, thus eliminating potential mistakes by programmers.

Another potential use case for our proposal is the ability for a translator to suggest possible improvements to code. If a translator determines that a block of code might benefit from execution on a quantum computer, a suggestion (similar to a warning) may be provided, allowing the programmer to consider alternative approaches. This approach could coexist with the API model: programmers would utilize APIs for calling quantum functions while the translator would provide suggestions when it encounters code that it believes would benefit from being executed on a quantum computer. The programmer would then consider replacing his or her code with a call to the appropriate quantum API.

2 Background

Quantum computing presents a sea change when compared with conventional computing. A basic understanding of key quantum mechanical concepts related to quantum computing (superposition, entanglement, decoherence, etc.) provides useful background knowledge. The next few sections will provide insight into various aspects of quantum computing. Topics directly related to quantum mechanics and how they affect the implementation of quantum computers will be discussed, as well as theoretical computer science.

It is worth noting that quantum mechanics can be difficult to grasp due to the fact that we do not experience quantum mechanical phenomena in everyday life. Only at atomic scales does quantum behavior make itself apparent.

2.1 Quantum Mechanical Concepts

2.1.1 Superposition

We generally think of waves and matter as being separate topics. For example, a book is comprised of particles, while the light emitted by a light bulb consists of of waves. But at the subatomic level, electrons and photons exhibit both particle-like and wave-like characteristics. Performing a measurement of a quantum system will result in "particle-like" behavior, i.e. a single state will be observed. Though when not disturbed by its external environment (e.g. taking a measurement of the system), the quantum system exhibits wave-like behavior, where the system may be said to exist as a combination of multiple states.

2.1.2 Entanglement

An entangled state is a state of a composite system whose subsystems are not probabilistically independent [3]. In other words, from an information-theoretic point of view, the information regarding an entangled system (e.g. a set of qubits) is not encoded locally in each particle (e.g. qubit), but rather in the correlation of the two [13]. This concept can be difficult to grasp, as this phenomena occurs at the scale of subatomic particles and

atoms. We typically do not see these effects in our macroscopic world. For our purposes, it will more than suffice to simply be familiar with the term.

2.1.3 Interference

Two waves may combine with each other, with the resultant wave's amplitude larger or smaller than the two original waves. The former is an example of constructive interference, while the latter is an example of destructive interference.

Algorithms designed to run on quantum computers rely on interference for finding solutions to problems that would take much longer to run on a conventional computer. Without the use of interference, a quantum algorithm may not even yield a correct solution. Interference may be the only phenomenon that matters for the analysis of quantum algorithms [28]. Other phenomena such as entanglement are important for the design of quantum computers, but do not carry much importance from the quantum algorithm designer's point of view [28].

2.2 Quantum Computing Basics

2.2.1 What is a Quantum Computer?

Today's conventional computing architecture is based on the concept of the bit: a unit of information that exists in *only* one of two possible states. These bits are typically represented as voltage levels in a computer's transistors, with each transistor containing one, or possibly more, bits (i.e. voltage states). For now, we will stick to the single-bit example. The key point here is that a transistor may be set to only one voltage at any given time. In more abstract terms, a value of 0 or 1. This concept forms the building block that conventional computers are based upon. If we assume n transistors in a computer, then at any given time the computer holds *one* of 2^n possible states.

In contrast to conventional computers, the building block of quantum computers is the quantum bit, or qubit. The key difference between a bit and a qubit is the latter's ability to exhibit superposition and entanglement. A qubit may exist as a superposition of its basis states (referred to here as 0 and 1), meaning that the qubit's state is a superposition of both 0 and 1. In this state, the qubit exhibits the wave-like behavior mentioned earlier. Maintaining a qubit in a state of superposition requires a high degree of isolation from its

external environment; the slightest amount of interference (e.g. from a photon or magnetic field) leads to a "collapse of the wave function", i.e. the qubit reverts to one of its basis states (*which* state is determined by some probability).

The use of entanglement is a second key factor differentiating conventional computing and quantum computing. A system of entangled qubits existing in states of superposition allows for the possibility of greatly increased computational power relative to conventional computers. If all n qubits in a system are entangled with each other, the number of possible states a quantum computer can hold at a given time is equal to 2^n (not *one* of 2^n states as with our conventional computing example, but all 2^n states *simultaneously*).

The details of how qubits are physically implemented in a quantum computer are beyond the scope of this thesis. Without delving too deeply into the topic, it is sufficient to say that these states can be represented by the spin states of an atom or the polarization of a photon [9]. The conventional equivalent being the states of voltage inside a transistor. In both cases the physical implementation is typically abstracted away by thinking of bits or qubits in terms of 0s and 1s.

2.2.2 Noise

Noise is possibly the biggest obstacle scientists face in the development of more powerful quantum computers. Superposition and entanglement make it possible for a quantum computer to attain its superior computing power relative to conventional computers (for certain problems). But maintaining qubits in states of superposition and entanglement requires keeping these qubits isolated from their external environment. A stray electromagnetic field is enough to cause a qubit to revert to one of its basis states.

2.2.3 Performing a Computation

The act of performing an observation is the final, and of course necessary, step in the process of performing a computation. After all, the observation is what tells us what the result of the computation is. But what all needs to happen before this?

The architecture of a quantum computer represents a fairly drastic change in comparison to the architecture of conventional computers. The computational logic of a quantum computer is implemented via gates, similar to conventional computers. But unlike conventional computers, certain quantum logic gates are specific only to the realm of quantum

computing.

The 2^n states in a quantum computer alluded to earlier provide significant potential for computing power, but without a properly implemented algorithm (e.g. an algorithm that finds the prime factors of an integer), the correct answer will not be found. To understand how this works, consider again the example of a single qubit in a superposition of two basis states, 0 and 1. It is at this point that the state of the qubit can be described as a wave function, existing as a combination of 0 and 1. Upon performing a measurement, the qubit's wave function collapses and the qubit assumes a state of 0 or 1 (due to the perturbations introduced by making the measurement). Which state it assumes is dependant upon a probability, where this probability is determined by the method used to place the qubit into a superposition of these basis states. Thus, the power behind quantum computing is not observable. We only see the result of this action.

It is a misconception that a quantum computer can try all 2^n possible answers and somehow provide the correct one. While the quantum computer does indeed hold these states simultaneously, the "magic" stops there. It is up to the algorithm to manipulate these states in order to find the correct answer to whatever problem is being solved. A fundamental part of quantum computing consists of utilizing constructive and destructive interference while performing a computation. (Recall that qubits existing as a superposition of states can be described by a wave function. The use of interference allows us to manipulate this wave function.) If the algorithm does what it is supposed to do, incorrect results will cancel out via destructive interference while the use of constructive interference will yield the desired answer upon performing a measurement. Without the use of interference, a measurement will yield a state randomly, rendering the quantum computer essentially useless.

2.2.4 Programming a Quantum Computer

One could imagine simply utilizing existing programming languages to program quantum computers. While this would be convenient, differences in the architecture of quantum computers and conventional computers mean that programming the former with a present-day programming language presents complications. Consider the no-cloning theorem, which forbids the creation of identical copies of arbitrary quantum states. Simply setting a variable equal to another variable would violate this principle (e.g. `x = y`, where `x` and `y` correspond to quantum registers comprised of qubits).

Present-day programming methods for quantum computers involve the use of languages specifically designed for quantum computers. This is done at a very low level by today's software standards, where the programmer manipulates individual gates and thus must possess knowledge of the underlying architecture. While this may suffice for the present-day primitive state of quantum computing, in a future where quantum computers are as ubiquitous as today's computers, a higher-level approach would be useful. After all, many programmers today need only a minimal amount of knowledge regarding the hardware their code runs on. Contrast this with the early days of conventional computing, where programming involved literally re-wiring the device (such as ENIAC).

It is our view that programmers of the future will benefit from an abstraction between the quantum hardware and the software they write, similar to the relationship between today's programmers and the hardware their code runs on. We explore potential ways to meet that goal in this thesis.

Many programming tasks are perfectly suited for a conventional computer, due to the fact that a quantum computer would provide no noticeable performance increase (we discuss this in more detail later). A future where both paradigms coexist with each other seems probable.

2.3 Computer Science Theory

Quantum computing may be seen by some as a sort of panacea, introducing a vastly more powerful form of computing that will render today's computers obsolete. Our current understanding of the topic suggests a more sober expectation is needed. While this new form of computing will entail significantly more processing power for particular types of problems, other tasks will see no speedup at all. One useful way to understand this issue is to apply concepts from computer science, such as algorithm analysis and computational complexity theory.

2.3.1 Algorithm Analysis

An algorithm is a computational procedure that takes a value, or set of values, as input, and produces some value, or set of values, as output [6]. An algorithm's performance is measured in terms of the amount of resources (e.g. time) it takes for the algorithm to be executed.

Computer scientists utilize several terms to describe the runtime characteristics of an algorithm. For our purposes, we will use what is known as "Big O notation", which provides an upper bound on a function, to within a constant factor [6]. In more practical terms, this notation describes an algorithm's runtime in terms of its input size. For example, an algorithm exhibiting linear runtime will see its runtime grow in proportion to its input size n: $O(n)$. In other words, if the input size doubles, so does the time it takes for the algorithm to run.

Consider a linked list containing n elements. Retrieving the first value from this list always takes the same amount of time, regardless of the size of the list. This task is said to run in constant time: $O(1)$. A task or algorithm running in constant-time is ideal, as the input size is irrelevant with regard to how long it takes the task or algorithm to complete. Unfortunately, only simple tasks can achieve such an ideal runtime.

Other algorithms may run in quadratic time, implying a runtime proportional to the square of the input size. Several notable sorting algorithms have quadratic runtimes.

2.3.2 Complexity Classes

Having gone over the method in which algorithm performance is measured, we can now look at grouping together algorithms with similar runtimes into what are called complexity classes.

The runtimes discussed above (constant [$O(1)$], linear [$O(n)$], and quadratic [$O(n^2)$]) are all part of the complexity class known as **P** ("Polynomial-time Find"). **P** is considered to be comprised of problems that can be solved efficiently. In other words, problems for which solutions can be found efficiently [11]. Computer scientists consider an efficient algorithm to be one that runs in polynomial time (i.e. the algorithm is in **P**), meaning that the runtime of the algorithm is of the form $O(n^c)$, where c is a constant. An inefficient algorithm would thus exhibit a worse runtime, for example, exponential runtime ($O(2^n)$), and thus not be in **P**.

Class **NP** is another important complexity class to consider. **NP** is defined to be the class of problems that have efficiently verifiable proof systems [11]. In more simplistic terms, **NP** is comprised of problems that can be verified in polynomial time, but not necessarily solved in polynomial time. For example, given an integer and the task of finding its prime factors, one may require quite a bit of time to find a correct solution. But if one were given both the integer to be factored plus a set of candidate prime factors with the task of

verifying whether or not the solution is correct, it would simply be a matter of multiplying these candidate prime factors together to determine if they are indeed prime factors of the original integer.

Verifying whether or not a solution is correct takes at most the same amount of time as actually solving a problem (and typically less time), so it stands to reason that all problems in **P** are in **NP**, with the latter likely containing more problems than the former (i.e. problems verifiable in polynomial time are not necessarily solvable in polynomial time). This, however, has not been proven, and is one of the most sought-after proofs in computer science.

2.3.3 Complexity Classes in Relation to Quantum Computing

How do these complexity classes fit into the realm of quantum computing? After all, quantum computing presents an entirely different approach to computing, and it stands to reason that what might be considered difficult or impossible using conventional computers might be easy with a quantum computer. Unfortunately, the answer is not what many might hope.

Our understanding of quantum computing is an evolving process. New research provides additional insight into the field and broadens our understanding of what may or may not be possible. It is important to consider that our present-day understanding of the field may, and likely will, change as research progresses. With this said, we can discuss currently held beliefs regarding the capabilities of quantum computers.

Figure 2.1 illustrates several complexity classes as many computer scientists believe them to exist (though as previously noted, this has not been proven, and there exists the possibility that **P** equals **NP** as well as **PSPACE**). These complexity classes are specific to today's computers. The realm of quantum computing has its own complexity classes, including **BQP** (Bounded-error Quantum Polynomial time), which is essentially the quantum equivalent of **P** (using a more technical definition: **BQP** is the class of all computational problems that can be solved efficiently on a quantum computer, where a bounded probability of error is allowed [21]).

PSPACE is a space analog to **P**, meaning that a problem is in **PSPACE** if it can be solved using a polynomial amount of space (e.g. memory). Like **P** and **NP**, it is believed that **P** and **PSPACE** are not equal, though this has not been proven. Exactly where **BQP** fits with respect to **P**, **NP**, and **PSPACE** is currently an open question [21].

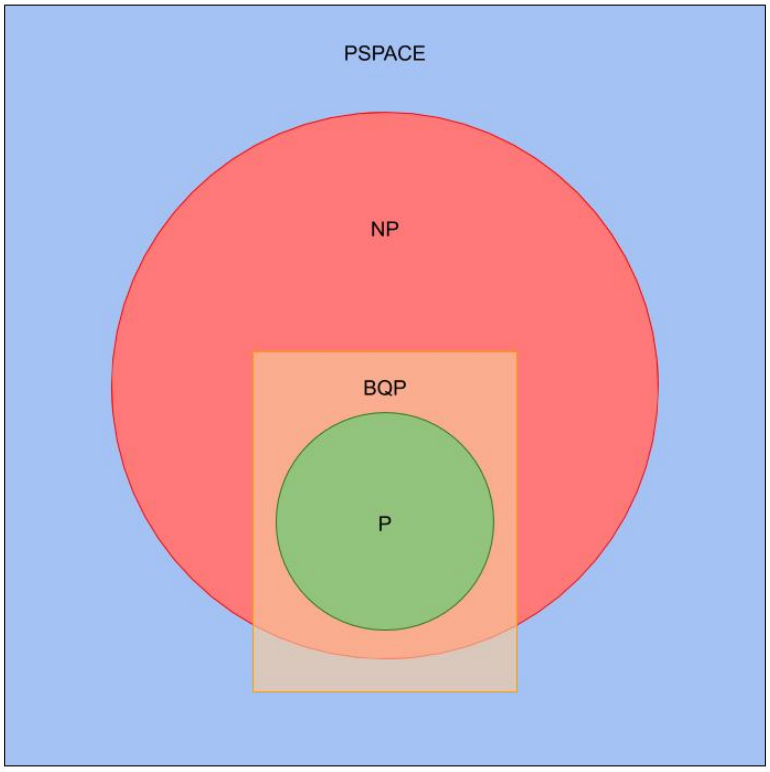

Figure 2.1: Depiction of relationship between **BQP** and other complexity classes as currently theorized.

Current theory suggests that quantum computers have the capability to solve certain **NP** problems in polynomial time. In addition, recent (May 2018) research shows that quantum computers may be able to *solve* problems in polynomial time that conventional computers cannot even *verify* in polynomial time [24]. In this case, class **BQP** extends beyond **NP** (as shown in Figure 2.1).

Seeing as how complexity classes **P** and **NP** are still not yet fully understood, it makes sense that complexity classes specific to quantum computing (e.g. **BQP**) are even less understood. Raz and Tal provide evidence [24] that there is likely much to still be learned regarding the capabilities of quantum computers.

Note the relationship between the various classes shown in Figure 2.1. **BQP** does not fully encompass **NP**, yet it also protrudes outside of **NP**. This implies that for some problems, a quantum computer provides no useful speedup in comparison to a conventional computer, while in other cases, a quantum computer provides a potentially massive speedup. Just how large is this advantage over conventional computers? Put another way, what would Figure 2.1 look like if it were drawn to scale? How many problems could a quantum computer efficiently solve that a conventional computer could not efficiently solve? Or

even more tantalizing: how many problems can a quantum computer efficiently solve that a conventional computer could not even efficiently verify the answer to (represented in Figure 2.1 as the region of **BQP** not contained in **NP**)? These questions are still being researched. A list of known quantum algorithms and their speedup relative to conventional computers [23] shows a decent number of problems with varying speedups, ranging from polynomial to exponential.

The advantages of quantum computing over its conventional counterpart are rooted in the assumption that there exist problems for which the former provides a (possibly significant) speedup relative to the latter. This set, or class, of problems can be visualized in Figure 2.1 as the region of **BQP** outside of **P** (where **BQP** is the class of problems solvable in polynomial time on a quantum computer, and **P** is the class of problems solvable in polynomial time on a conventional computer).

Certain problems are said to be **BQP**-complete, meaning that these problems are the hardest in **BQP**. The study of **BQP**-complete problems enhances our understanding of **BQP**, similar to how the study of **NP**-complete problems enhances the understanding of non-determinism [29]. As a side note, every problem in **NP** can be transformed in polynomial time into a problem in **NP**-complete. In addition, every **NP**-complete problem is also **NP**-hard, where **NP**-hard is the set of problems that are at least as hard as the hardest problems in **NP**. Thus, proving that a single **NP**-complete problem can be solved in polynomial time would mean that $\mathbf{P} = \mathbf{NP}$. **BQP** has a similarly high level of importance to the realm of quantum computing. And similar to **NP**-completeness and its relationship to **NP**, the fact that a problem is **BQP**-complete means that any problem in **BQP** can be transformed into that problem.

2.4 Code Translation

The code translation process is the ideal point at which recognition of quantum-compatible code would take place. It is thus useful to touch upon the basics of this process.

While a compiler and interpreter are generally viewed as performing two separate tasks, they can both be viewed as translators [25], where code in one form is translated into another form. A compiler typically translates a higher-level language source program into an equivalent target program, such as assembly language or machine language. This process involves a thorough analysis of the code, with the end result being a stand-alone target program. This target program can then be executed later. Execution of a compiled

program does not involve the compiler, as the program was already compiled at some previous point in time.

In contrast, an interpreter performs translation and execution at the same time. Statements are read and executed one by one, with no thorough analysis of the code as with a compiler. As a general rule of thumb, a language is interpreted if the translator is "simple", while a compiled language requires a "complicated" translator [25]. Put another way, compilation involves thorough analysis and nontrivial transformation [25].

Figure 2.2 and Figure 2.3 provide illustrations of compilation and interpretation.

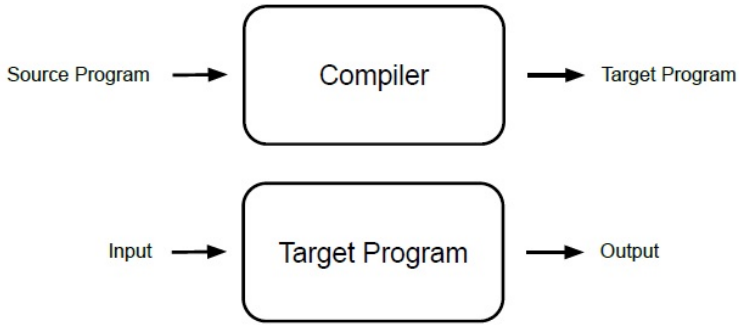

Figure 2.2: Compiler High-Level View.

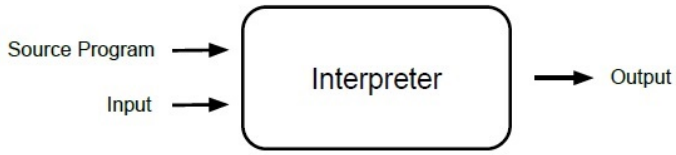

Figure 2.3: Interpreter High-Level View.

Compilation is a multi-step process with various elements contributing to the compilation of a program. A typical compiler may perform some or all of the following tasks during compilation [1] [25]:

1. Lexical Analysis: Performed by the scanner. During lexical analysis (or "scanning"), individual characters composing the source program are read and grouped into sequences known as tokens, which are passed on to the next phase of compilation.

2. Syntax Analysis: Performed by the parser. During syntax analysis (or "parsing"), the token stream received from the scanner is parsed and an intermediate representation of code is created, typically in the form of a so-called parse tree. Syntax errors

are caught during parsing, e.g. a variable name that doesn't conform to the rules of the language.

3. Semantic Analysis: Semantic analysis is the process of uncovering the meaning of a program. Using the intermediate code representation generated by the parser, the semantic analyzer typically builds and maintains a symbol table data structure that maps each identifier to the information known about it. In addition, higher-level semantic rules of the language are enforced at this stage, such as type checking and properly defined subroutine calls.

4. Intermediate Code Generation: A compiler may generate a low-level intermediate interpretation at this stage of compilation. This form may then be mapped to the target language later on.

5. Machine-Independent Code Optimizer: A variety of optimizations may take place, such as those concerning speed, size, or even power usage.

6. Code Generator: The intermediate representation generated earlier is mapped to the target language.

7. Machine-Dependent Code Optimizer: Additional optimization may take place based on the target machine.

The gate-level method of programming modern-day quantum computers does not require this translation process, as the source code is already at the level of machine code. Depending on how quantum computing progresses, the time may come when a more high-level method of programming quantum computers becomes worthwhile. One possible method is to utilize today's programming techniques by merging the conventional and quantum worlds together and rely on the code translation process to decide what is executed on conventional hardware and what is executed on quantum hardware. This method would essentially be an extension of what is used today, with only a slight (though decidedly non-trivial) modification needed to the previously described compilation process. This modification involves the recognition of code that the translator deems suitable for execution on a quantum computer, via program equivalence checking.

2.5 Program Equivalence

Program equivalence is a relevant topic in computer science with numerous applications, including algorithm recognition, program verification, program optimization, and compiler optimization [15]. The term "semantic equivalence" may be used to describe programs that are equivalent: even if written differently, they have the same meaning. For example, a particular sorting algorithm may be implemented in various ways, but (assuming they are implemented correctly) still implement the same algorithm and thus would be considered semantically equivalent. Program equivalence concerns semantic equivalence as it relates to programs.

The "equivalence problem" asks whether it is possible to determine if two programs are equivalent. The answer, as it relates to "decidability", is that it varies, based on the representation (e.g. context-free grammar, finite-state automata, etc.) of the programs being compared. A problem is called *decidable* if there exists a "yes" or "no" answer to a given problem [19].

The equivalence problem is said to be undecidable "as soon as the considered program class is rich enough to be interesting" [15]. It has been formally shown that the equivalence problem is decidable in simple cases [12]. Other research into program equivalence has shown promising results [5] [26] [17] [4]. Simple cases of algorithm recognition can be solved using pattern matching [2].

The ability to recognize a particular algorithm has a direct implication on our aforementioned translation process whereby quantum code and conventional code are discerned. Our testing will demonstrate that it is indeed possible to compare two pieces of code and determine if they are semantically equivalent.

3 Research Methods and Methodology

3.1 Research Methodology

The idea of utilizing program equivalence checking in order to identify quantum-compatible code is an unexplored topic. However, as already discussed, program equivalence itself is a well-known and researched topic [5] [26] [17] [4].

Our goal is to propose a method that utilizes program equivalence checking for the automatic recognition of code suitable for execution on a quantum computer. This entails the ability to analyze code and determine the presence of an algorithm that should be executed on a quantum computer. Program equivalence checking is the key to this process. We thus needed a software tool that can compare two pieces of code and determine whether or not they are semantically equivalent by means of program equivalence checking.

While there exist various program verification tools, many of these focus on a single source file, whereby verification according to a certain specification is performed on this code. Our proposal centers on the ability to compare *two* pieces of code, meaning that we needed a software tool that performs program verification checking on both pieces of code and determines if there exists semantic equivalence between them. This process would thus simulate our comparison between a benchmark and arbitrary code.

A paper on a novel approach to program equivalence checking [18] along with an implementation of the technique has direct applications to our own research. The authors' program equivalence checking algorithm compares a block of code with another block of code and determines if they are semantically equivalent. The ability to perform program equivalence checking directly relates to our own work: the identification of code that has been determined to be a "conventional" equivalent of a quantum algorithm, i.e. an algorithm designed for execution on a quantum computer.

According to the authors, many verification tools (including several that we considered using) are unable to prove equivalence of most programs containing loops [18]. On the other hand, their program equivalence checking algorithm is applicable to programs con-

taining zero or more (nested) loops [18]. This fact further justified our decision to use their software tool for our research.

3.2 CORK

The name of this software tool is CORK (Compiler Optimization coRrectness checKer), created by Nuno P. Lopes and Jose Monteiro. CORK is the implementation of the compiler verification technique proposed in the authors' paper [18]. While compiler optimization is different from our intended goal, it is nonetheless related to our work. Rather than the optimization of code, our goal is algorithm recognition. However, the CORK tool has strong relevance to our proposal in that it verifies whether or not two pieces of code are semantically equivalent. CORK performs analysis on code written in a simplistic language created by the authors (what they call "WHILE"). Consequently, we cannot apply CORK for code written in a programming language such as C or Java. Instead, we will translate a particular block of code that we want to perform analysis on into the WHILE language. The "arbitrary" code that we want to compare to this benchmark will undergo the same translation procedure. This concept is illustrated in Figure 3.1 (discussed in more detail in Section 3.3). The goal will be to work with simple chunks of code in order to eliminate problems associated with our code translation.

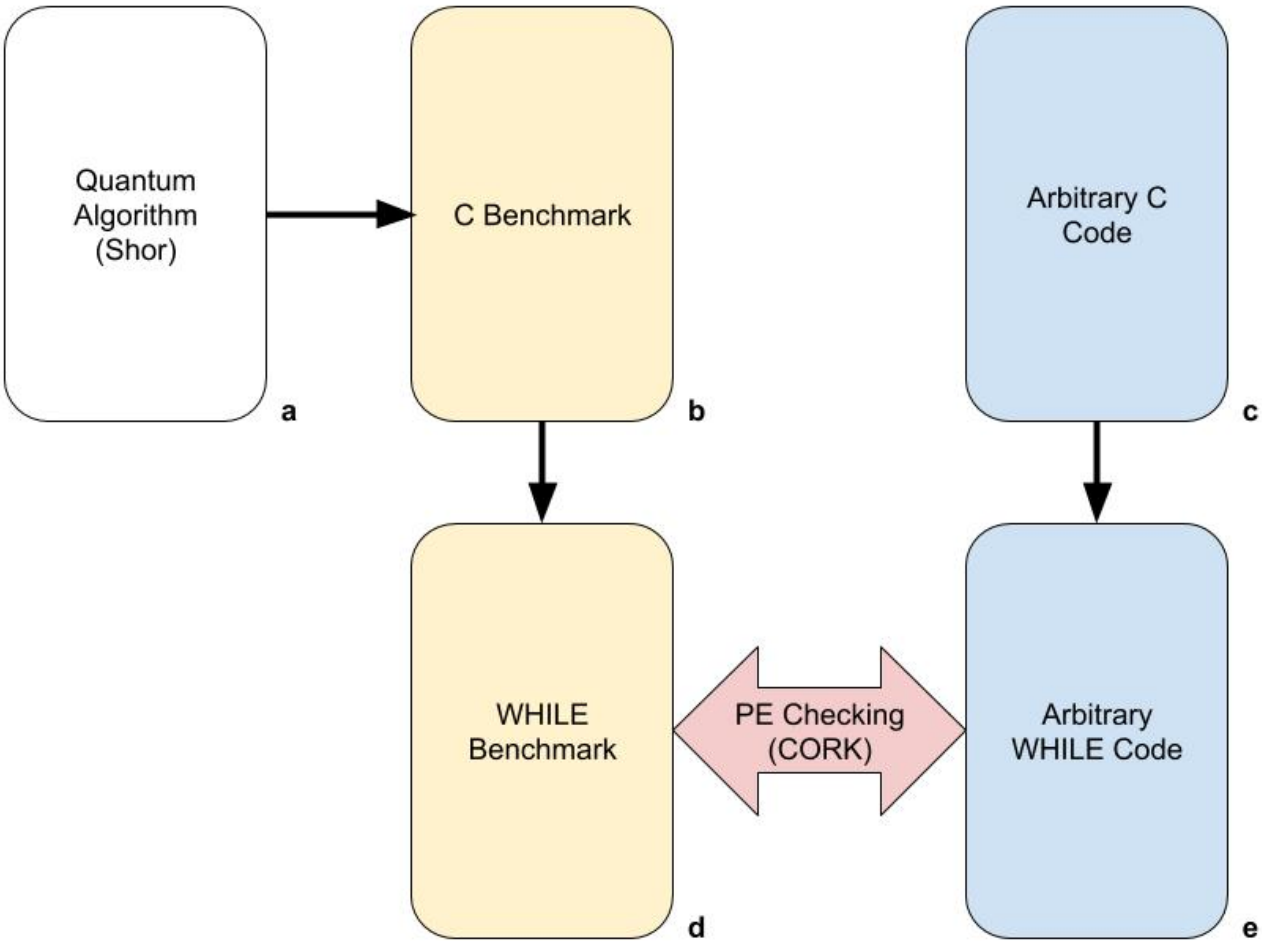

Figure 3.1: The program equivalence checking procedure performed in this thesis. The repeat period-finding procedure of Shor's algorithm (**a**) is implemented via the C programming language and serves as the benchmark (**b**). The benchmark is manually translated into the language used by CORK, known as "WHILE" (**d**). A modified version of the benchmark is created (**c**), intended to be either semantically equivalent or semantically inequivalent to the benchmark. This code is also manually translated into WHILE (**e**). CORK is thus used to perform program equivalence checking between the two WHILE scripts, with a determination made regarding semantic equivalence.

Performing a test with CORK requires three files:

1. .orig file: The original block of code written in WHILE. This file serves as our benchmark.

2. .opt file: The so-called optimized block of code written in WHILE. Our "arbitrary

code" will be implemented via this file throughout our testing.

3. .wp file: A listing of template statements, where each template statement entry describes input parameters along with parameters modified by the function. A template statement is a placeholder for a variable assignment, function call, or other loop that may be present in a loop under optimization [18]. (The authors' own examples include template statements outside of loops, which is why we use template statements outside of loops as well.) Each template statement includes a *read set* and a *write set*, which can be thought of as lists of inputs and outputs, respectively. These lists contain variables explicitly used in the script, as well as context variables. Context variables represent the effects of a template statement S, though they are not explicitly used in the script. In general, the read and write sets of each template statement must contain at least one context variable.

Wolfram Mathematica is utilized by CORK for constraint and recurrence solving.

Figure 3.2 provides the WHILE language syntax along with a short description (both taken directly from [18]). UF stands for Uninterpreted Function. UFs are frequently used in software verification tasks [18] and are appealing because they allow for irrelevant details of programs to be abstracted out [18].

$$
\begin{aligned}
e &::= n \mid v \mid e_1 \oplus e_2 \mid \mathsf{UF}(e_1, \ldots, e_n) \\
b &::= e \leq 0 \mid b_1 \otimes b_2 \\
c &::= \textbf{skip} \mid v := e \mid c_1 \,;\, c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c_1 \mid \textbf{assume } b \\
&\quad \mid \textbf{assert } b
\end{aligned}
$$

Figure 3.2: WHILE language syntax. n is an integer number, v is a variable name, UF is an uninterpreted function symbol, $\oplus$ is a binary operator over integer expressions (e.g. $+$, $-$), and $\otimes$ is a binary operator over a boolean expression (e.g. $\wedge$, $\vee$).

3.3 Research Methods

Our translation process is illustrated in Figure 3.1. We start with a block of code that we designate to be the benchmark. Considering the theme of this thesis, this block of code should have high relevance to the realm of quantum computing, such as a quantum algorithm. We chose a partial implementation of Shor's algorithm as found in [16], using the C programming language. As the book describes, this particular block of code is the conventional implementation of an algorithm that finds a function's repeat period. While

the details are beyond the scope of this thesis, it shall suffice to say that determining a function's repeat period is part of Shor's algorithm. The book provides a nice description of Shor's algorithm, including the repeat period-finding procedure used here.

With the benchmark determined, our next step is to write programs that will test the program equivalence checking technique performed by CORK. The authors of CORK provide some inspiration with regard to our testing approach. Along with CORK itself, the authors provide examples of various tests they performed by means of test scripts executed by CORK. These tests involve verification of simulated compiler optimizations, such as loop peeling and loop unrolling. The authors applied these various compiler optimizations to non-optimized scripts and then used CORK to determine if the two scripts (the non-optimized/original script and the optimized script) were semantically equivalent. Our testing follows a similar approach, in that we begin with a block of code (the repeat period-finding procedure of Shor's algorithm) to serve as the original script (i.e. benchmark) and write various modified versions of this script via compiler optimizations to serve as the arbitrary code. CORK is used for performing program equivalence checking between the benchmark and arbitrary code.

Some versions of code we write will be intentionally semantically inequivalent to the benchmark in order to test if CORK recognizes this difference. Others will be various compiler optimizations that are intended to be equivalent. We verify the functionality of the various implementations of C code by printing the repeat period as calculated by the algorithm.

The "Quantum Algorithm" of Figure 3.1 is the repeat period-finding procedure of Shor's algorithm in our case. This implementation (using the C programming language) serves as the "C Benchmark" shown as **b**. For the purposes of testing with CORK, we translate this C code into CORK's WHILE language (box **d**). We use CORK to compare this WHILE benchmark against other WHILE code ("Arbitrary WHILE Code", box **e**), with a determination regarding semantic equivalence provided by CORK. Thus, for our testing purposes, program equivalence checking takes places between boxes **d** and **e** of Figure 3.1.

A real-world implementation of this proposal would of course exclude boxes **d** and **e** of Figure 3.1, and would instead follow the process described in Figure 1.2. The program equivalence checking process would be performed by our hypothetical translator as depicted in Figure 1.1.

We translate the C code implementations into WHILE scripts, with the benchmark and arbitrary code versions corresponding to .orig and .opt files, respectively.

Several important points need to be made regarding the WHILE language:

1. Variable names are of the form `V#`, where `#` is an integer.
2. For-loops are not supported, hence why for-loops in the C code implementations are converted into while-loops in the WHILE implementations.
3. Function calls (or even functions themselves) are not supported.
4. The modulus operator is not supported.
5. The return operator is not supported.

Due to these limitations, the process of translating C code to WHILE code involves replacing certain lines of code with template statements (a template statement is of the form `S#`, where `#` is an integer). For example, line 4 of Listing 4.4 is a placeholder for line 15 of Listing 4.2. Line 15 of Listing 4.2 contains both a function call and an assignment. For our purposes, the entire line is abstracted into a template statement. Our use of template statements can be seen as abstractions needed for the translation of C code to WHILE code. While not ideal, they nonetheless allow us to proceed with our testing.

It should be noted that the term "optimized" is used due to the intended use of CORK: verifying compiler optimizations. This differs from our idea, where the second block of code is not an optimization: it is simply code that may or may not be semantically equivalent to the benchmark.

Our test environment is as follows:

- Ubuntu 18.04.3 LTS
- CORK v0.1
- GCC v7.5.0
- Wolfram Mathematica v8.0.4

4 Program Equivalence Checking with CORK

Various types of compiler optimizations were tested along with more simplistic code modifications. Figure 4.1 illustrates the data provided for each test we perform. For each test, we show the C code and its corresponding WHILE equivalent (manually translated from the C version). For each of these versions, we provide both the benchmark and modified version (the modified version corresponds to the "arbitrary code" being checked against the benchmark in Figure 3.1). The benchmark is the same throughout all tests. We list it next to the various modified versions for convenience. We also list the output of both the modified C version (the calculated repeat period) along with the output of CORK (whether or not CORK found the two scripts to be semantically equivalent).

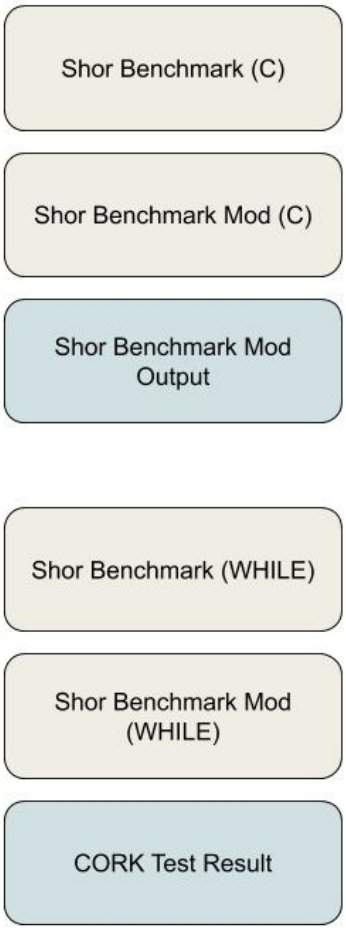

Figure 4.1: For each test, the following data is provided: the repeat period-finding procedure of Shor's algorithm ("Shor Benchmark") along with the modified version of this file that serves as the arbitrary code being compared to the benchmark (both the C and WHILE versions of these two files are listed). In addition, the output of the modified Shor repeat period-finding code is provided (i.e. the calculated repeat period) as well as the output of CORK's program equivalence analysis of the two WHILE files.

Figure 4.2 shows our repeat period-finding benchmark implemented as C code and its equivalent WHILE implementation.

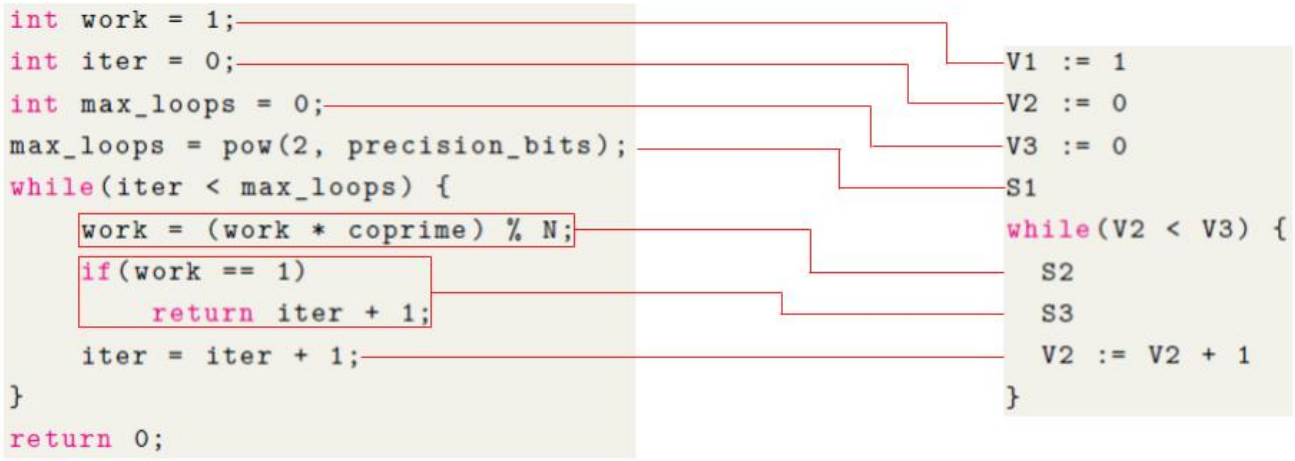

Figure 4.2: The repeat period-finding benchmark used in our testing. Shown here is the C code version and its equivalent WHILE implementation. The red lines link corresponding C code and WHILE code. For example, variable `work` in C code is implemented as variable `V1` in WHILE code. Due to limitations associated with WHILE, certain lines of code are substituted with template statements (i.e. `S1`, `S2`, and `S3`).

Listing 4.1 shows the C version of ShorNoQPU translated from the authors' pseudocode [16]. Due to the aforementioned limitations of WHILE (i.e. no for-loops, etc.), we modified this version into something compatible with WHILE, shown in Listing 4.2. The for-loop is replaced with a while-loop and variables are initialized before being used. This modified version of ShorNoQPU defines the C benchmark (lines 11 through 23 of Listing 4.2). The corresponding WHILE benchmark is shown in Listing 4.4. We will list the C and WHILE benchmarks throughout our testing results for ease of reference.

For the sake of clarity, it is worth noting again that the C and WHILE versions of the benchmark are specific only to our testing. The need to manually translate the benchmark to another language (C to WHILE) is due to our test environment. A real-world implementation would not include this procedure.

Listing 4.3 shows the expected output of the C benchmark (repeat period of 12).

Seven tests were performed, with five producing expected results as reported by CORK. Two tests were written such that they were intended to be semantically inequivalent to the benchmark. CORK correctly identified these as such (i.e. they were part of the five expected results).

Changes to the modified versions with respect to the benchmark are highlighted via color-coded text. Red text indicates code that is moved or duplicated. Blue text indicates

new code. Moved or duplicated code is marked in both the benchmark and the modified versions for ease of identification.

```c
#include <stdio.h>
#include <math.h>

int ShorNoQPU(int N, int precision_bits, int coprime);

int main(void) {
    int repeat_period = ShorNoQPU(35, 4, 2);
    printf("repeat period = %d\n", repeat_period);
}

int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int max_loops = pow(2, precision_bits);
    for(int iter = 0; iter < max_loops; ++iter) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
    }
    return 0;
}
```

Listing 4.1: Repeat period-finding procedure (using the C programming language) as listed in [16].

```c
#include <stdio.h>
#include <math.h>

int ShorNoQPU(int N, int precision_bits, int coprime);

int main(void) {
    int repeat_period = ShorNoQPU(35, 4, 2);
    printf("repeat period = %d\n", repeat_period);
}

int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.2: Code in Listing 4.1 converted to format compatible with WHILE (i.e. for-loop converted to while-loop and variables initialized before being used). Function ShorNoQPU defines the benchmark.

```
repeat period = 12
```

Listing 4.3: C Benchmark Output

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.4: WHILE Benchmark

4.1 No Modification

This test was simply meant to show that CORK identifies duplicate code as being semantically equivalent.

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.5: C Benchmark

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.6: C No Modification

```
repeat period = 12
```

Listing 4.7: C No Modification Output

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.8: WHILE Benchmark

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.9: WHILE No Modification

```
Optimization is correct
```

Listing 4.10: CORK No Modification Analysis Result

4.1.1 Test Analysis

CORK identified duplicated code as being semantically equivalent. No modification was made to the second version, i.e. Listing 4.5 and Listing 4.6 are identical (along with their corresponding WHILE versions, Listing 4.8 and Listing 4.9). CORK's analysis showed that the two versions are indeed semantically equivalent.

4.2 Loop Modification I

CORK correctly identified semantic inequivalence between the benchmark and modified version as shown in Listing 4.16. Listing 4.13 displays the incorrectly calculated repeat period.

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.11: C Benchmark

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        iter = iter + 1;
        if(work == 1)
            return iter + 1;
    }
    return 0;
}
```

Listing 4.12: C Loop Modification I

```
repeat period = 13
```

Listing 4.13: C Loop Modification I Output

```
1 V1 := 1
2 V2 := 0
3 V3 := 0
4 S1
5 while(V2 < V3) {
6   S2
7   S3
8   V2 := V2 + 1
9 }
```

Listing 4.14: WHILE Benchmark

```
1 V1 := 1
2 V2 := 0
3 V3 := 0
4 S1
5 while(V2 < V3) {
6   S2
7   V2 := V2 + 1
8   S3
9 }
```

Listing 4.15: WHILE Loop Modification I

```
1 FAILED to prove path equivalence
```

Listing 4.16: CORK Loop Modification I Analysis Result

4.2.1 Test Analysis

The while-loop of the WHILE Benchmark consists of two template statements followed by an increment of the loop counter. Listing 4.15 shows the modified version, where the loop counter `iter` is incremented before S3 is executed. We see the corresponding C code in Listing 4.12. By incrementing the loop counter before the if-statement, program behavior will differ (compare lines 8 through 10 of Listing 4.11 and Listing 4.12). Variable `iter` will have a value one greater than that of the benchmark when `work == 1` and execution returns via line 10.

4.3 Loop Modification II

Similar to the previous test, CORK correctly identified semantic inequivalence between the benchmark and modified versions as shown in Listing 4.22. Listing 4.19 displays the incorrectly calculated repeat period.

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.17: C Benchmark

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        iter = iter + 1;
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
    }
    return 0;
}
```

Listing 4.18: C Loop Modification II

```
repeat period = 13
```

Listing 4.19: C Loop Modification II Output

```
1 V1 := 1
2 V2 := 0
3 V3 := 0
4 S1
5 while(V2 < V3) {
6   S2
7   S3
8   V2 := V2 + 1
9 }
```

Listing 4.20: WHILE Benchmark

```
1 V1 := 1
2 V2 := 0
3 V3 := 0
4 S1
5 while(V2 < V3) {
6   V2 := V2 + 1
7   S2
8   S3
9 }
```

Listing 4.21: WHILE Loop Modification II

```
1 FAILED to prove path equivalence
```

Listing 4.22: CORK Loop Modification II Analysis Result

4.3.1 Test Analysis

Similar to the previous test, this test also involved modifying the location of the loop counter's incrementation. This time, the loop counter was moved to the top of the while-loop, i.e. lines 7 and 8 of Listing 4.12 were swapped, yielding Listing 4.18. Doing so did not affect the calculated repeat period (relative to the previous test). However, the repeat period was still incorrectly calculated for the same reason as in the previous test. CORK correctly identified semantic inequivalence here as well.

4.4 Loop Peeling

A correct repeat period of 12 was calculated by the code shown in Listing 4.25. CORK verified Listing 4.27 as being equivalent to the benchmark, i.e. the two WHILE scripts are semantically equivalent.

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.23: C Benchmark

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    if(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.24: C Loop Peeling

```
repeat period = 12
```

Listing 4.25: C Loop Peeling Output

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.26: WHILE Benchmark

```
V1 := 1
V2 := 0
V3 := 0
S1
if(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.27: WHILE Loop Peeling

```
Optimization is correct
```

Listing 4.28: CORK Loop Peeling Analysis Result

4.4.1 Test Analysis

Loop peeling is a compiler optimization technique where one or more iterations of a loop are removed and performed outside of the loop body in order to improve efficiency. Listing 4.24 shows our "loop peeling" implementation (the reason for the quotations will be explained briefly). Lines 6 through 11 have been duplicated, though instead of a while-loop, this block consists of an if-statement.

Now for why "loop peeling" is in quotations above: this is not a practical implementation of loop peeling. A more realistic use of loop peeling would involve the replacement of certain variables with hardcoded values, due to the fact that this code is executed at most once. We know what the values of these variables will be, so there is no reason to have the CPU perform unnecessary load operations from memory. Thus, we would replace `iter` with `0` on line 6, replace line 9 with `return 1;`, and replace line 10 with `iter = 1;`. Variable `work` to the right of the assignment operator on line 7 can be removed.

The reason why we didn't implement this in Listing 4.24 is due to the fact that making these modifications would result in different code, even if it is the same in a semantic sense. The use of hardcoded values would simply replicate the first iteration of the while-loop (in a more efficient manner, hence the point of using loop peeling in the first place), meaning that a program equivalence analyzer should find this version of code semantically equivalent with the benchmark. Listing 4.27 is the WHILE equivalent of Listing 4.24. At lines 6 and 11 of Listing 4.27 is `S2` (corresponding to lines 7 and 13 of Listing 4.24). At lines 7 and 12 of Listing 4.27 is `S3` (corresponding to lines 8-9 and 14-15 of Listing 4.24). Each instance of `S2` and `S3` corresponds to identical code. By implementing loop peeling, `S2` on line 6 and `S3` on line 7 would no longer be written the same as `S2` on line 11 and `S3` on line 12. We could *assume* that they're the same, considering their semantic similarities, but the point here is to have CORK decide if they're the same, and to minimize the number of assumptions we need to make. WHILE does not support the modulus operator or return statement, which is why we utilize template statements (i.e. UFs) for these particular lines. Were we able to use the modulus operator and return statement, we would have been able to fully implement loop peeling. This would also have allowed us to eliminate the use of `S2` and `S3` in all of our tests as well.

Listing 4.24 is thus a partial implementation of loop peeling that doesn't actually provide any performance gain the way it is currently written. On the other hand, considering that our focus is program equivalence checking, not compiler optimization, our loop peeling test still serves a useful purpose. CORK confirmed semantic equivalence between the benchmark (Listing 4.26) and Listing 4.27, which is what we expected.

4.5 Loop Tilting

This test was one of two where CORK produced unexpected results. CORK reported Listing 4.33 as not being semantically equivalent to the benchmark, while the C version (Listing 4.30) produced the expected repeat period of 12.

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.29: C Benchmark

```
int ShorNoQPU(int N, int precision_bits, int coprime, int B) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        int iter2 = 0;
        while(iter2 < B) {
            work = (work * coprime) % N;
            if(work == 1)
                return (iter + iter2) + 1;
            iter2 = iter2 + 1;
        }
        iter = iter + B;
    }
    return 0;
}
```

Listing 4.30: C Loop Tilting

```
repeat period = 12
```

Listing 4.31: C Loop Tilting Output

```
1 V1 := 1
2 V2 := 0
3 V3 := 0
4 S1
5 while(V2 < V3) {
6   S2
7   S3
8   V2 := V2 + 1
9 }
```

Listing 4.32: WHILE Benchmark

```
1 V1 := 1
2 V2 := 0
3 V3 := 0
4 S1
5 while(V2 < V3) {
6   V7 := 0
7   while(V7 < V8) {
8     S2
9     S3
10    V7 := V7 + 1
11  }
12  V2 := V2 + V8
13 }
```

Listing 4.33: WHILE Loop Tilting

```
1 FAILED to prove path equivalence
```

Listing 4.34: CORK Loop Tilting Analysis Result

4.5.1 Test Analysis

Loop tilting involves grouping loop iterations together in order to take advantage of caching. Listing 4.30 illustrates this technique. The inner while-loop performs up to B iterations, where B is the maximum number of loop iterations allowing for data to stay in a cache line. This is useful if the loop body contains operations performed on an array, with each iteration modifying elements of the array.

In our case, loop tilting would not be of any use, as lines 9 through 12 involve memory accesses at the same addresses. However, the idea of this thesis centers around program equivalence, so even if the compiler optimization applied here is not very practical, it still

presents a useful test case for our purposes. We also need to remember that a programmer may write code in any number of ways, some of better quality than others. This brings us back to the point of this thesis, which is to investigate the potential to compare two blocks of code and determine if they are semantically equivalent. There is no telling how an arbitrary block of code will be written. Two pieces of code may differ significantly yet be semantically equivalent.

We suspect that the reason for CORK not finding Listing 4.32 (the benchmark) and Listing 4.33 semantically equivalent is due to the loop condition variable `V3` (Listing 4.33, line 5) being determined by template statement `S1` (Listing 4.33, line 4). If we modify the aforementioned .wp file that determines each template statement's inputs and outputs such that the value of `V3` is no longer determined by `S1`, CORK finds the two scripts to be semantically equivalent.

The problem here appears to be the fact that `V3` is determined by a UF. One restriction of CORK is that loop conditions may not involve "UF applications", where, as previously described, UF stands for Uninterpreted Function. A template statement is an example of a UF. The WHILE benchmark and several other CORK scripts illustrate that the output of a UF evidently can be used as a loop condition, yet based on this particular test, it appears that using the output of `S1` as part of the loop condition for the if-statement on line 5 is the cause of CORK not finding the scripts to be semantically equivalent. It could be that the manner in which the script is written is part of the problem: the while-loop nested within the if-statement is different from other test scripts we have written. This, in combination with `S1` modifying `V3`, might be tripping up CORK.

4.6 Loop Unrolling

The C version correctly produced a repeat period of 12, and CORK reported semantic equivalence of the corresponding WHILE scripts.

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.35: C Benchmark

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter + 1 < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;

        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }

    if(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.36: C Loop Unrolling

```
repeat period = 12
```

Listing 4.37: C Loop Unrolling Output

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.38: WHILE Benchmark

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 + 1 < V3) {
  S2
  S3
  V2 := V2 + 1
  S2
  S3
  V2 := V2 + 1
}

if(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.39: WHILE Loop Unrolling

```
Optimization is correct
```

Listing 4.40: CORK Loop Unrolling Analysis Result

4.6.1 Test Analysis

Loop unrolling involves reducing the total number of loops performed by duplicating code in the loop body. The result is a decrease in overhead by means of reducing the number of increments performed on the loop counter variable, with a drawback being an increase in code size. The additional code associated with ensuring the extra code stays within the

intended number of loops may itself present an increase in overhead. Duplicating the code of multiple loops (instead of just two, as in our test) may be required in order to account for this extra overhead.

Lines 12 through 15 of Listing 4.36 contain a duplicate of lines 7 through 10. By executing this code twice, we halve the number of loop counter incrementations needed in the loop, which might be useful for a speed-critical application.

4.7 Software Pipelining

This test was the second instance of an unexpected result, with CORK reporting semantic inequivalence. The C version produced a correct repeat period of 12.

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.41: C Benchmark

```
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    if(iter < max_loops) {
        work = (work * coprime) % N;

        while(iter < max_loops - 1) {
            if(work == 1)
                return iter + 1;
            iter = iter + 1;
            work = (work * coprime) % N;
        }

        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

Listing 4.42: C Software Pipelining

```
repeat period = 12
```

Listing 4.43: C Software Pipelining Output

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

Listing 4.44: WHILE Benchmark

```
V1 := 1
V2 := 0
V3 := 0
S1
if(V2 < V3) {
  S2
  while(V2 < V3 - 1) {
    S3
    V2 := V2 + 1
    S2
  }
  S3
  V2 := V2 + 1
}
```

Listing 4.45: WHILE Software Pipelining

```
FAILED to prove path equivalence
```

Listing 4.46: CORK Software Pipelining Analysis Result

4.7.1 Test Analysis

Software pipelining is an assembly-level optimization technique whereby the compiler rearranges instructions as needed in order to avoid dependencies and make more efficient use of the processor's pipeline. We followed the code example presented by the authors of CORK and applied the idea to our C code version (Listing 4.42). We realize that this does not have much practical use: the technique is typically applied to assembly code,

not to a higher-level language such as C. Our example assumes that the programmer had a good reason to arrange the code as is; perhaps execution is improved when written in this manner. Or, it could be that this is simply how the programmer chose to write his or her code. Again, we are investigating the concept of program equivalence. The use of compiler optimizations in this thesis is a means to an end, not the end itself. While software pipelining is an interesting concept, the important point here is that we have a block of code to compare to the benchmark, even if it is not a straightforward example of software pipelining.

In Listing 4.45 we see that template statement `S3` is not executed immediately after template statement `S2`, unlike in the benchmark. The hypothetical programmer may have an explicit reason for doing so, such as for performance (though again, in this particular example, there would likely be no benefit to arranging the code in this manner). Or, as previously mentioned, it could be that this is simply how the programmer wrote the code, even if it is needlessly convoluted. Programmers rarely write perfect code (if ever), so this particular bit of code can be thought of as an example of imperfect code that still does what it is intended to do (though CORK evidently disagrees).

Similar to our loop tilting test in Section 4.5, this test also failed. We believe the cause of the failures is related: the use of a UF application's output as a loop condition. Modifying the .wp file such that `S1` no longer writes the value of `V3` results in CORK finding the two scripts to be semantically equivalent (similar to our loop tilting test). In both cases, using `V1` as the loop condition would not be very useful. The point here is that using a non-UF variable as the loop condition results in the test passing, implying that the issue seems to be the fact that the loop condition is determined by a UF output.

5 Discussion

5.1 Summary of Results

Test	Repeat Period	Expected Result?	CORK	Expected Result?
No Modification	12	Yes	Equivalent	Yes
Loop Modification I	13	Yes	Not Equivalent	Yes
Loop Modification II	13	Yes	Not Equivalent	Yes
Loop Peeling	12	Yes	Equivalent	Yes
Loop Tilting	12	Yes	Not Equivalent	No
Loop Unrolling	12	Yes	Equivalent	Yes
Software Pipelining	12	Yes	Not Equivalent	No

Figure 5.1: The calculated repeat period and CORK analysis result is listed for each test. A "Yes" or "No" accompanies each field, indicating whether or not the result was expected.

5.2 Research Questions

We will now evaluate the extent to which our two research questions were answered.

5.2.1 What methods can be used to automatically identify code suitable for execution on a quantum computer?

CORK is a software tool that verifies the correctness of compiler optimizations. Seeing as how the verification of compiler optimizations relates to our goal of program equivalence checking, we chose to use CORK as a means of performing program equivalence checking during our testing. Our findings indicate that the compiler verification capability presented by CORK is a step toward our envisioned ability to automatically identify quantum-compatible code from an arbitrary block of code. Given a benchmark, CORK was able to identify semantically equivalent code in a highly-controlled environment. These tests were

trivial and in no way constitute a real-life environment. However, based on our limited testing, we believe that a base exists for further development, as compiler verification is a related, existing branch of computer science.

If the technique proposed by this thesis is to become a reality, a significant leap from existing capabilities will be required. CORK is able to perform program equivalence checking only when the following conditions have been met:

1. The scripts corresponding to the so-called "original" code and "optimized" code have been written in the WHILE language. For our testing, this entailed a manual translation from the C programming language to CORK's WHILE language.

2. A separate file (.wp file) containing the template statements associated with these two files along with their read and write sets is present.

3. Adherence to the strict constraints of the WHILE language (i.e. for-loops, return statements, the modulus operator, and functions themselves are not supported). A WHILE script is essentially a simple block of code. Higher-level constructs such as functions are beyond the scope of WHILE.

Our hypothetical translator must be able to compare the benchmark to all portions of code in the files being compiled. Unlike the highly controlled environment used in our testing, the arbitrary code being compared against the benchmark in a real-life environment would be the entire compilation unit. Determining program equivalence in simple cases is manageable. More complex cases, such as those likely to be encountered in real-world scenarios would potentially be much more difficult, even *undecidable* [30]. This presents a potential limitation to our proposal which will be addressed when we discuss our second research question.

The process of performing program equivalence checking would likely require the use of a compiler. It is difficult to envision an interpreted language allowing for the requisite analysis of code in order to determine what should be executed on a quantum computer instead of a conventional computer. One possible exception would be the use of a Just-In-Time compiler that analyzes code before it is executed by an interpreter.

Focusing only on compilation, it is worth exploring in a bit more detail the process a compiler would take in analyzing code. As mentioned in Section 1, compilation is a multi-step process. Determining if code is suitable for quantum hardware requires an understanding of the semantics of said code. Referring back to our description of the

typical steps associated with compilation, it would make sense to perform this analysis during the semantic analysis phase. Figure 5.2 illustrates the compilation process laid out earlier.

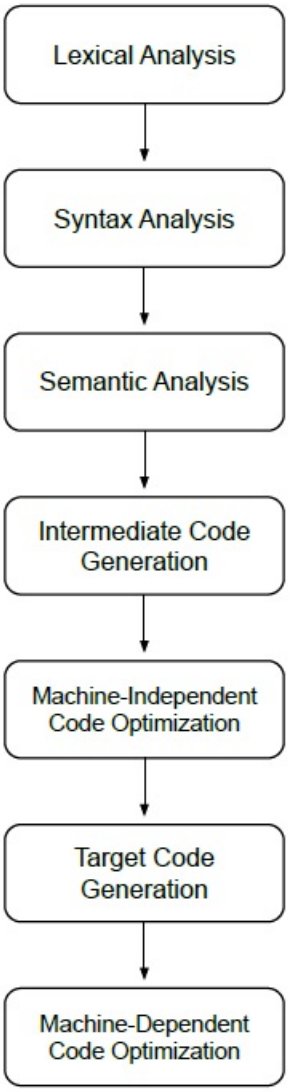

Figure 5.2: Compilation Steps

It is during the syntax analysis phase that a compiler constructs an internal representation of the code structure, typically via a parse tree [1]. The semantic analysis phase utilizes this parse tree in order to check the source program for semantic consistency with the language definition [1]. In addition to the parse tree, the symbol table generated and utilized by the semantic analyzer makes the semantic analysis phase a good place to perform program equivalence checking of quantum-compatible code, as the semantic analyzer has the ability to analyze and understand the semantics of the program.

Determining that a block of code should be executed on a quantum computer is not enough; the code then actually needs to be executed on a quantum computer. Using Shor's algorithm as an example, upon recognition of the algorithm by the compiler, the code would be removed and replaced with a request to a quantum computer instructing it to run Shor's algorithm with a given set of parameters, dependent upon the programmer's code. At runtime, Shor's algorithm is executed on a quantum computer.

The location of this quantum computer might be in the same device as the conventional computer or in the cloud. The latter is more probable in the initial stages of quantum computing. Perhaps some day we will advance to a point where quantum processing units are as ubiquitous as their conventional, modern-day counterparts, and are compact enough to be placed in handheld devices.

We should also consider the possibility that a cloud-based quantum environment could have issues at runtime. Consider the following: several sections of code suitable for execution on a quantum computer are found during compilation. These sections are replaced with some yet-to-be-determined method of contacting a cloud-based quantum computer with a request to execute a particular quantum algorithm. Of course, this is still compile-time. The compiled program (and its associated quantum portion) won't be executed until later. Suppose that during runtime, a problem occurs and the device the code is executing on is unable to contact a quantum computer. This would be a problem if a quantum target was the only option for the code to run on. Perhaps it would be worthwhile to compile this code for a conventional target as well, to account for this potential issue.

To summarize, automatic recognition of code suitable for execution on a quantum computer requires the use of program equivalence checking. Compiler verification is a well-studied field that has implications for this topic. And as we have shown in our testing, the ability to compare two pieces of code and determine if they are semantically equivalent is feasible.

5.2.2 What are the limitations of these methods' capabilities?

The equivalence problem places a theoretical limitation on the extent to which our quantum code identifying proposal can perform program equivalence checking. However, the capabilities of program equivalence checking are advancing [5] [26] [17] [4], and simple instances of algorithm recognition can be solved using pattern matching [2]. It has been formally shown that the equivalence problem is decidable in simple cases [12].

We showed in our testing that it is possible to conclude that two programs are semantically equivalent in certain, albeit trivial, tests. The need to translate C code to WHILE code, the limitations of the WHILE language, etc. all imply that this is hardly a real-world example. Our testing should be viewed more as a proof of concept instead of an example of a real-world program equivalence checking method.

Our tests did produce two unexpected results, but the ability to correctly determine semantic equivalence or semantic inequivalence was nonetheless established for multiple tests. We believe that the cause of these failures (with a "failure" meaning that CORK reported semantic inequivalence between the two scripts when semantic equivalence was expected) was the use of a UF application's output as part of the loop condition. It is stated in [18] that loop conditions cannot involve UF applications. However, we use a UF application's output as part of the loop condition for our benchmark with no apparent adverse affects.

Only one of the two variables in the loop condition of the WHILE benchmark is the output of a UF application (`V3`). Further testing found that making both `V2` and `V3` the output of UF applications did not result in a fail. We tested this by taking a successful test (loop peeling) and adding a UF to write `V2`. CORK found the two scripts to be semantically equivalent. Thus, in spite of the fact that both variables of the loop condition are the output of UF applications, CORK was still able to find the scripts semantically equivalent, as expected.

While this additional testing indicates that UF application output usage alone does not have an adverse effect on CORK's ability to perform program equivalence checking, the two tests that failed appear to be partially the result of using `V3` as the output of a UF application. Our testing showed that a combination of using `V3` as the output of a UF application along with nested loops (e.g. the loop tilting test in Section 4.5) or a while-loop nested inside an if-statement (e.g. the software pipelining test in Section 4.7) resulted in CORK reporting semantic inequivalence even though semantic equivalence was expected. However, the absence of one or both of these conditions results in expected behavior: the WHILE benchmark demonstrates that the use of a UF application's output as a loop condition is not a problem. In addition, the nested blocks used in the loop tilting and software pipelining tests pose no problem as long as a UF application's output is not used as a loop condition.

Our discussion with the authors of CORK lead us to believe that these failures are unexpected with regard to CORK's theoretical capabilities. The authors also point out that different versions of Wolfram Mathematica may have an effect on test results. The upshot

being that the two failures may have been influenced by an unexpected issue with CORK and/or Wolfram Mathematica. In theory, the ability to perform program equivalence checking on these two pairs of scripts (and report semantic equivalence) is not beyond the capabilities of CORK.

5.3 C to WHILE Translation

We wanted to use a common, high-level programming language in our testing, and decided upon the C programming language. With CORK utilizing its own WHILE language, a manual translation of C code to WHILE code was necessary in order to proceed with testing. Once this translation was complete and we had "original" and "optimized" WHILE scripts at hand, it was possible to perform testing.

A real-world implementation would need a more automated approach. As discussed earlier, a compiler implementing our program equivalence checking proposal could perform program equivalence checking during the semantic analysis phase of compilation. Interpretation would likely require a Just-In-Time compiler in order to perform the necessary analysis needed to identify quantum-compatible code.

5.4 Size of BQP

Considering the importance of **BQP**-completeness, the discovery of new **BQP**-complete problems [29] and advancement in the understanding of **BQP** [24] represents important progress for quantum computing. Unfortunately, it appears as though progress on the development of quantum algorithms is lagging behind the progress being made on quantum computing hardware. Peter Shor, author of Shor's algorithm, provides two possible reasons for this trend. The first is that there may only be a handful of problems for which quantum computers offer a substantial speedup over conventional computers, and we have already discovered most of the algorithms pertaining to these problems [27]. The second possibility is that quantum computing presents such a drastic change from its conventional counterpart that we simply have not attained the necessary level of understanding needed to progress more rapidly [27].

The range of problems in **BQP** has an impact on the usefulness of our automatic program equivalence checking proposal. If there are indeed only a handful of problems that are

worthy of execution on quantum computers, it would reduce the need for automatic program equivalence checking and make an API-based approach more practical. The need for a programmer to have familiarity of only several algorithms, that are used in very specific cases, is not a lot to ask for. Should the number of problems in **BQP** grow, then our idea begins to increase in usefulness.

5.5 Rebuttal to the Quantum API Argument

A potential argument against the idea proposed in this thesis is the fact that APIs are in existence today, and are commonplace among software development. A quick check of the Java Platform API Specification shows just how large this collection of APIs is. Many, if not most, programming languages utilize APIs. Their usage allows for programmers to make a call to a specific library instead of writing the code themselves, eliminating redundancy by providing a more efficient way of writing code. Of course, properly utilizing an API requires knowledge of said API, and the larger an API is, the more is required of the programmer to understand the potentially large number of libraries the API provides access to. A programmer may not realize a library call already exists, and write the code himself/herself, resulting in wasted time, and a potentially less efficient implementation.

Considering the current number of known problems in **BQP**, the use of a quantum API makes sense. The potential for a programmer to not be aware of a particular algorithm's presence in this API is lessened. Additionally, certain algorithms may be specific to the quantum-world, meaning that an implementation via conventional programming techniques is impossible. However, as shown by our testing with Shor's algorithm, certain scenarios may allow for an implementation on either conventional or quantum hardware. And while forgoing the use of an API by means of a programmer's own implementation may result in slightly reduced performance on a conventional computer, the same situation in a quantum environment could result in a drastic reduction in performance, relative to what would be achieved with quantum hardware. In other words, if a programmer does not realize that a quantum API exists and writes an implementation for a conventional computer instead of calling the equivalent quantum API, the loss in performance could be enormous. This is possibly the main attraction to using the automated approach suggested in this thesis.

An increase in the number of problems in **BQP** makes the automatic approach more attractive. Even if the quantum API model wins out, the use of program equivalence

checking techniques could still be of use by means of suggestions to the programmer if the compiler (or some other software application such as an Integrated Development Environment) believes that code written by the programmer could be replaced by a quantum API call.

6 Conclusion

The idea presented in this thesis is the automatic recognition of code suitable for execution on a quantum computer. Instead of relying on the programmer to decide whether code should be executed on a quantum computer or a conventional computer, the goal here is to remove the decision altogether from the programmer and automate the process. Our testing presents a step in this direction. Using CORK, a software tool that implements a novel program equivalence checking algorithm, we were able to perform program equivalence checking on two blocks of code and determine if there exists semantic equivalence between the two. This was repeated for multiple successful tests. This process would thus be utilized for the purpose proposed in this thesis: the automatic recognition of code suitable for execution on a quantum computer.

Our idea is dependent upon several key assumptions: the field of quantum computing advancing far beyond present-day capabilities, a programming environment where quantum code and conventional code are interspersed with each other, and (expanding upon the previous assumption) a preference for the quantum and conventional realms to be abstracted away from the programmer, as opposed to an API model where the programmer explicitly decides when use of a quantum computer is necessary. An additional assumption is a sufficiently large number of problems in **BQP** that would make an automatic approach to quantum code recognition useful.

Program equivalence checking has important applications in computer science, and thus has attracted interest by means of research within the computer science community. The results of this research can be seen in various research papers and software tools, such as CORK, utilized in our own testing. A base for further progress thus exists, and with the likely progression of quantum computing, automatic recognition of quantum-compatible code could be of use in the future.

Further progress must be made regarding quantum hardware. Problems with noise need to be overcome in order for quantum computers to perform increasingly non-trivial tasks. On the software end, the number of problems that present a significant speedup relative to conventional computers needs to increase. This entails progress made within the computer science community regarding quantum algorithm research. Finally, the field of program equivalence plays an essential role with regard to the idea proposed in this thesis. The

ability to analyze code via program equivalence checking in a real-world environment and make a determination regarding semantic equivalence is quite a major step compared to the testing performed in this thesis. In order for our idea to progress, research into program equivalence is vital.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson Education, Inc., 2007. ISBN: 0-321-49169-6.

[2] C. Alias and D. Barthou. "On the Recognition of Algorithm Templates". In: *Electronic Notes in Theoretical Computer Science* 82.2 (2003).

[3] G. Auletta, M. Fortunato, and G. Parisi. *Quantum Mechanics*. Cambridge University Press, 2009, p. 183. ISBN: 978-0-511-65042-0.

[4] B. R. Churchill. *Blackbox Equivalence Checking of Program Optimizations*. 2019.

[5] B. Churchill, O. Padon, R. Sharma, and A. Aiken. "Semantic Program Alignment for Equivalence Checking". In: PLDI 2019 (2019), pp. 1027–1040. DOI: 10.1145/3314221.3314596. URL: https://doi.org/10.1145/3314221.3314596.

[6] T. H. Cormen. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 978-0-262-03384-8.

[7] *D-Wave Systems*. Accessed 26 July 2020. URL: https://www.dwavesys.com.

[8] D. Deutsch. "Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer". In: *Proceedings of the Royal Society of London* 400.1818 (1985), pp. 97–117.

[9] D. P. DiVincenzo. *The Physical Implementation of Quantum Computation*. 2008, p. 2. DOI: arXiv:quant-ph/0002077.

[10] R. Feynman. *Simulating Physics with Computers*. 1981. DOI: https://doi.org/10.1007/BF02650179.

[11] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. 1st ed. Cambridge University Press, 2008, p. 59. ISBN: 0-321-49169-6.

[12] T. Harju and J. Karhumäki. "The Equivalence Problem of Multitape Finite Automata". In: *Theor. Comput. Sci.* 78.2 (1991), pp. 347–355. DOI: 10.1016/0304-3975(91)90356-7. URL: https://doi.org/10.1016/0304-3975(91)90356-7.

[13] J. D. Hidary. *Quantum Computing*. Springer, 2019, p. 7. ISBN: 978-3-030-23922-0.

[14] *IBM Research - Quantum*. Accessed 26 July 2020. URL: http://www.research.ibm.com/quantum.

[15] G. Iooss, C. Alias, and S. Rajopadhye. “On Program Equivalence with Reductions”. In: (Sept. 2014). DOI: 10.1007/978-3-319-10936-7_11.

[16] E. R. Johnston, N. Harrigan, and M. Gimeno-Segovia. *Programming Quantum Computers: Essential Algorithms and Code Samples*. 1st ed. O’Reilly Media, Incorporated, 2019. ISBN: 978-1-492-03968-6.

[17] S. Kundu, Z. Tatlock, and S. Lerner. “Proving Optimizations Correct using Parameterized Program Equivalence”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 44 (May 2009), pp. 327–337. DOI: 10.1145/1542476.1542513.

[18] N. P. Lopes and J. Monteiro. *Automatic Equivalence Checking of UF+IA Programs*. 2013.

[19] M. Margenstern. “Frontier Between Decidability and Undecidability: A Survey”. In: *Theoretical Computer Science* 231.2 (2000), pp. 217–231.

[20] *Microsoft - Quantum*. Accessed 26 July 2020. URL: https://www.microsoft.com/en-us/quantum.

[21] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. 7th ed. Cambridge University Press, 2010, p. 41. ISBN: 978-1-107-00217-3.

[22] *Qiskit*. Accessed 26 July 2020. URL: https://qiskit.org.

[23] *Quantum Algorithm Zoo*. Accessed 30 August 2020. URL: http://quantumalgorithmzoo.org.

[24] R. Raz and A. Tal. “Oracle Separation of BQP and PH”. In: *Electronic Colloquium on Computational Complexity* 107 (2018).

[25] M. L. Scott. *Programming Language Pragmatics*. Academic Press, 2000. ISBN: 1-55860-442-1.

[26] K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. “Verification of Source Code Transformations by Program Equivalence Checking”. In: *Lecture Notes in Computer Science* 3443 (Apr. 2005), pp. 221–236. DOI: 10.1007/978-3-540-31985-6_15.

[27] P. W. Shor. “Progress in Quantum Algorithms”. In: *Quantum Information Processing* 3.1 (Oct. 2004), pp. 5–13. DOI: 10.1007/s11128-004-3878-2. URL: https://doi.org/10.1007/s11128-004-3878-2.

[28] J. Van Gael. *The Role of Interference and Entanglement in Quantum Computing*. 2005, p. 41.

[29] P. Wocjan and S. Zhang. "Several natural BQP-Complete problems". In: (July 2006).

[30] V. A. Zakharov. "The Equivalence Problem for Computational Models: Decidable and Undecidable Cases". In: *Third International Conference, MCU 2001* (Chișinău, Moldova). Ed. by M. Margenstern and Y. Rogozhin. Springer, Berlin, Heidelberg, May 2001, pp. 133–152. ISBN: 978-3-540-45132-7.